Compilation (#5) : Syntax-Directed Code Generation

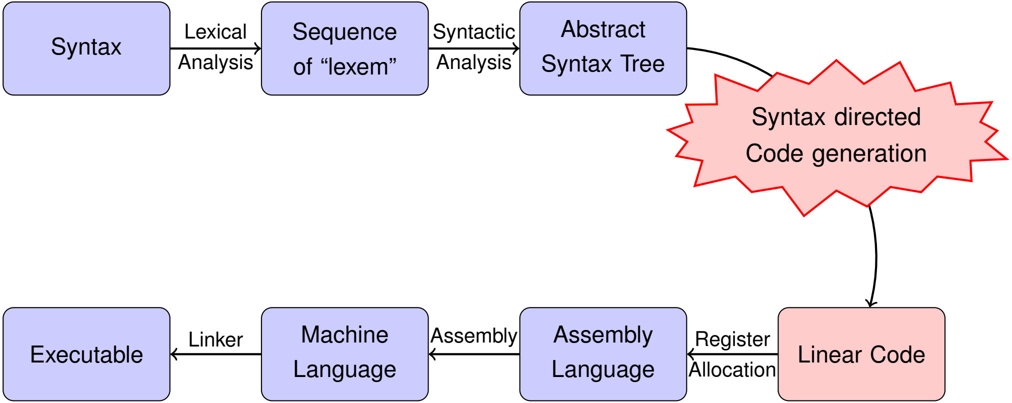
Laure Gonnord & Matthieu Moy & Gabriel Radanne & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

ENS DE LYON

Big Picture


Syntax
Lexical Analysis
Sequence of “lexem”
Syntactic Analysis
Abstract Syntax Tree
Syntax directed Code generation
Linear Code
Register Allocation
Assembly Language
Assembly
Machine Language
Linker
Executable

Rules of the Game here

For this code generation:

- Still no functions and no non-basic types. (MiniC w/o functions and strings)
- Syntax-directed: one grammar rule $\rightarrow$ a set of instructions. ▶ Code redundancy.
- No register reuse: each value is stored in its own location (potential memory waste)

The Target Machine: RISCV (course #1)

Code Generation vs Memory/Register Allocation

- Code generation in two steps:
  1. Generate instructions without deciding where data is stored (put everything it temporaries)
  2. Decide where each temporary is allocated (register? stack?)
- Temporary (sometimes called "virtual register"): temporary where data can be stored. Difference with (physical) registers:
  - They don't exist in the real processor / instruction set
  - There are an infinity of them

A first example (1/2)

How do we translate:

```
int x, y;
x=4;
y=12+x;
```

- Variable decl's visitor gives a temporary to each variable: $x \mapsto temp0$, $y \mapsto temp1$.
- Compute 4, store somewhere, then copy in x's temporary.
- Compute $12 + x$: 12 in `temp2`, copy the value of x in `temp3`, then add, store in `temp4`, then copy into y (i.e. `temp1`).

▶ Create temporaries whenever needed.

A first example: 3@code (2/2)

"Compute 4 and store in x (temp0)":

```
li temp2, 4
mv temp0, temp2
```

Objective

3-address RISCV **Code Generation** for the Mini-While language:

- All variables are int/bool.
- All variables are global.
- No functions (just `main()`).

with syntax-directed translation. Implementation in Lab (`MiniC/`)

▶ This is called **three-address code generation**

1. 3-address syntax-directed Code Generation
   - Rules

Code generation utility functions

We will use:

- A new (fresh) temporary can be created with a `fresh_tmp()` function.
- A new fresh label can be created with a `fresh_label()` function.
- The generated instructions are close to the RISCV ones.

Abstract Syntax

Expressions:

e	$::=$	c	constant
	$\mid$	x	variable
	$\mid$	$e + e$	addition
	$\mid$	e or e	boolean or
	$\mid$	$e < e$	less than
	$\mid$	...	

Statements:

S	$::=$	$x := expr$	assign
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	`if` b `then` S_1 `else` S_2	test
	$\mid$	`while` b `do` S `done`	loop

Code generation for expressions, example

e ::= c (cte expr)	`dest <- fresh_tmp()`
	`code.add("li dest, c")`
	`return` `dest`

- this rule gives a way to generate code for any constant.

Code generation for a boolean expression, example

e $::= e_1 < e_2$	`dest <- fresh_tmp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- fresh_label()`
	`code.add("li dest, 0")`
	`# if t1>=t2 jump to endrel`
	`code.add("bge endrel, t1, t2")`
	`code.add("li dest, 1")`
	`code.addLabel(endrel)`
	`return dest`

- integer value 0 or 1 to represent true/false.

Second example: a boolean test

Let us generate the code for $x < 4$ (assuming x is stored in `temp0`):

Second example: a boolean test

Let us generate the code for $x < 4$ (assuming x is stored in `temp0`):

```
li temp3, 4 // get 4
li temp2, 0
bge temp0, temp3, lbl0 // >= comp + jump
li temp2, 1
lbl0:
// Here, temp2 contains 1 if x<4, 0 otherwise
```

Code generation for commands, example

`if` b `then` $S1$ `else` $S2$	<pre>lelse <- fresh_label()
	lendif <- fresh_label()
	t1 <- GenCodeExpr(b)
	#if the condition is false, jump to **else**
	code.add("beq *lelse*, *t1*, 0")
	GenCodeSmt(S1) # then
	code.add("j *lendif*")
	code.addLabel(lelse)
	GenCodeSmt(S2) # **else**
	code.addLabel(lendif)</pre>

Example for `if/else`

Let us generate the code for `if (x<4) then y=7 else ...` (y in `temp1`)

Example for `if/else`

Let us generate the code for `if (x<4) then y=7 else ...` (y in `temp1`)

```
## code from previous slide here to compute x<4
beq temp2, zero, lelse1 // if false, jump
li temp4, 7
mv temp1, temp4 // y gets 7
j lendif1 // don't forget this one!
lelse1:
  # code for else branch
lendif1:
```

Example for `if/else`

Let us generate the code for `if (x<4) then y=7 else ...` (y in `temp1`)

```
## code from previous slide here to compute x<4
beq temp2, zero, lelse1 // if false, jump
li temp4, 7
mv temp1, temp4 // y gets 7
j lendif1 // don't forget this one!
lelse1:
  # code for else branch
lendif1:
```

Note: more efficient version possible by branching directly to `lendif1`/`lelse1` in the generation of $x < 4$ (get rid of `temp2`, save one beq), but this is sufficient for us.

From 3@ code to valid RISCV

3@code is not valid RISCV code!

We explore several allocation algorithms:

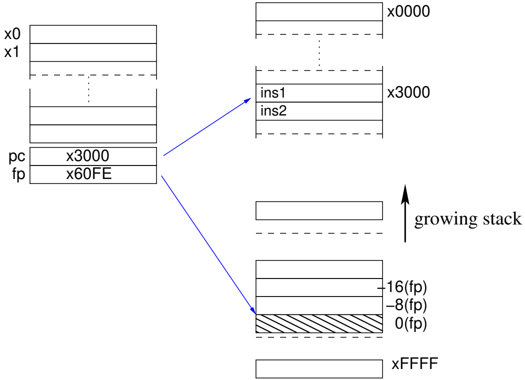
- All in registers $temp_i \rightarrow register$ $\leftarrow$ very, very naive
- All in memory $temp_i \rightarrow memory$ $\leftarrow$ very naive
- Store several temporaries in the same place, use registers and memory $\leftarrow$ yes, we'll do smart stuff too :-)

A stack, why?

- Store local variables of each functions
- Provide an easy way to communicate arguments values (see later)
- Give place to store intermediate values (e.g. `2*3` in `x = 1 + 2 * 3`)

Stack with RISCV

- Special register `fp` = Frame Pointer, points to the current stack frame.
- Store and loads from `fp`



Nice picture by N. Louvet - adapted in 2019

How to store into the stack

Store (the content of) s_3 on the stack at offset `offset`:

```
sd s3, -offset*8(fp)
# To generate from Python:
# sd(s3, Offset(FP, -offset*8))
# "write the value of s3 at address fp - offset*8"
```

Exercice: 3 address code generation for

```
i = 0;
if (i == 10) {
   i = i + 1;
} else {
   i = i - 1;
}
```

Exercice: naive allocation (all in registers)

```
    li temp_0, 42
    li temp_1, 1
    add temp_2, temp_1, temp_0

```

Exercice: “all in mem” allocation

```
    li temp_0, 42
    li temp_1, 1
    add temp_2, temp_1, temp_0

```

Code Generation

Input: a MiniC file:

```
int main() {
  int n;
  n=6;
  return 0;}
```

Output: a RISCV file:

```
[...]
        ;; (stat (assignment n = (expr (atom 6)) ;))
        li t1, 6    ; t1 is a riscv register.
        mv t2, t1
[...]
```

Steps

- 3-address code generation according to the code generation rules.
- Simple register/memory allocation + pretty print.

Details in the dedicated slides.

Drawbacks of this “all in mem” allocation

Drawbacks:

- Memory intensive loads and stores (each operation loads and store from memory)
- Uses a lot of memory (no reuse of memory for different computations)

Next:

- Store as many temporaries as possible in the same location
  $\rightsquigarrow$ Are two temporaries in conflict (i.e. used at the same time)?

▶ We need a more efficient data structure to reason on: **the control flow graph (CFG)**. (see next course)

Summary : 3 address code generation

1. 3-address syntax-directed Code Generation
   - Rules
2. Memory allocation
3. LAB: Direct Code Generation
4. Exercises
5. Conclusion